Machine Learning Engineer Nanodegree

Capstone Project

Frank Ding
2018

I. Definition

Project Overview

This capstone project is well defined by Udacity MLND Robot Motion Planning Capstone Project - Plot and Navigate a Virtual Maze whose original idea takes inspiration from Micromouse competitions, which originated in the late 1970s as an event where small robot mice solve a 16 x 16 maze. The autonomous micromouse robot is placed in a corner of the maze and is taksked to reach maze center. The mouse is given two runs in the maze. In the first run, it attempts to map out the maze to not only find the center, but also figure out the best paths to the center. In second run, the micromouse robot aims to reach the center in the fastest time possible.

Problem Statement

The goal is to design a algorithm that makes the micromouse robot find the center as quick as possible. To be most efficient includes not only the moves in the second run but also the moves in the first run, ie. explore the maze in an efficient way. More concretely, the score is defined as follows $Score = NumSteps_1/30 + NumSteps_2$

where $NumSteps_1$ is number of steps taken in first run and $NumSteps_2$ is number of steps in second run, capped by $NumSteps_1 < 1000, NumSteps_2 < 1000$

The rationale of the metric is further explained in Metrics section.

The guiding problem solving principles are

- In second run, calculate a shortest path using partial knowledge of the maze acquired in first run because trials in second run would not be economical since exploration incurs 1/30 cost in trial stage than in second stage.
- In first run, the robot is required to reach goal due to first principle. Since there is restriction on total number of steps and also number of steps in this stage has metric impact, it's wise for the robot to reach goal in minimal steps. This stage in first run is called **goal searching exploration** in this report.
- And if remaining steps allowed, the robot can explore other unexplored squares as many as possible in the maze in most efficient way in the hope to find shortcut path for next stage. This continuing exploration in first run can be considered as optimization so it is called **continuing exploration** in this report. It must make trade-off between the number

of step expected to be taken and number of steps that a promising shortcut path could reduce compared to existing shortest path.

Hence, the first principle would lead naturally to calculating shortest path using Dijkstra shortest path algorithm.

The second principle would make us find quickest paths while exploring using graph traveral algorithms such as A* search algorithms because aimless wondering would increase the possibility of failing to reach goal.

The third principle forces designers to prefer depth first search graph traversal algorithms than breadth first ones because breadth first search incurs more step cost due to back and forth movements. However, how to exploit existing knowledge of the maze and the effectiveness of making bet on moving to unexplored cells are unknown.

Metrics

The evaluation metric is listed below $Score = NumSteps_1/30 + NumSteps_2$

where $NumSteps_1$ is number of steps taken in first run and $NumSteps_2$ is number of steps in second run, capped by $NumSteps_1 < 1000$, $NumSteps_2 < 1000$

The total score not only considers total steps taken in test run but also involves steps in exploration run. Think about why it is necessary to include steps in exploration run. Suppose total score only considered step number in test run, i.e $Score = NumSteps_2$, robot is encouraged to take arbitrary time probing the maze and have a complete knowledge of the maze. Then in the second run, some deterministic shortest path algorithm can be employed to compute minimal steps. The overall result would be that every robot has the optimal score for a particular maze though they have wide range of steps probing the maze. Worst is that some robot may never terminiate in exploration run.

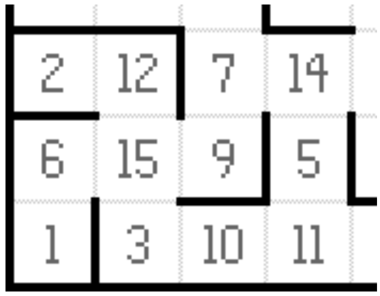
II. Analysis

Data Exploration

Udacity provides starter code including the following files:

- robot.py - This script establishes the robot class. This is the only script that you should be modifying, and the main script that you will be submitting with your project.
- maze.py - This script contains functions for constructing the maze and for checking for walls upon robot movement or sensing.
- tester.py - This script will be run to test the robot’s ability to navigate mazes.
- showmaze.py - This script can be used to create a visual demonstration of what a maze looks like.
- test_maze_##.txt - These files provide three sample mazes upon which to test your robot.

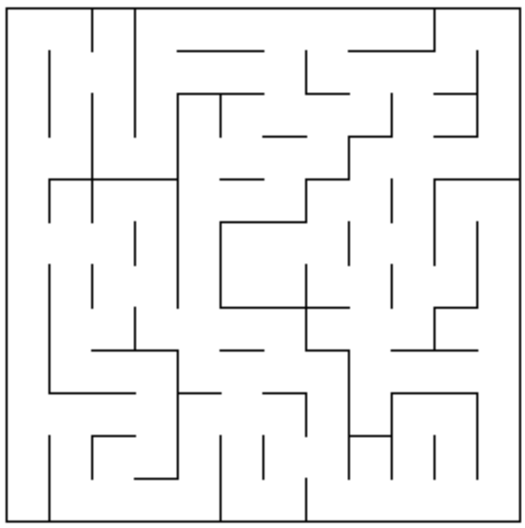
Maze is inputed as a text file. On the first line of the text file is a number describing the number of squares on each dimension of the maze n. On the following n lines, there will be n comma-delimited numbers describing which edges of the square are open to movement. Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom (0*1 + 1*2 + 0*4 + 1*8 = 10). Note that, due to array indexing, the first data row in the text file corresponds with the leftmost column in the maze, its first element being the starting square (bottom-left) corner of the maze.



Exploratory Visualization

Maze 1 (12x12)

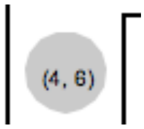
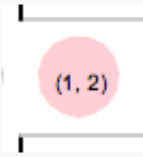
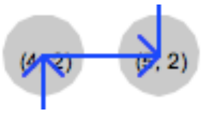
Maze visualization can be produced by *show_maze.py* in starter code. Below is the picture illustrating maze 1.



I enhanced visualization script in *vis.py*, with the following enhancements:

- cell coordinate is displayed
- cell status can be easily spotted by its color
- current cell together with its heading is denoted by red arrow
- route can be visualized as well

Legend Explained

Legend	Meaning
	fully explored cell walls on left and right edges top and bottom directions connect neighbouring cells
	not fully explored cell unknown status of top and bottom directions
	goal cell
	current cell with heading
	route with direction and step

Maze 1 is drawn below

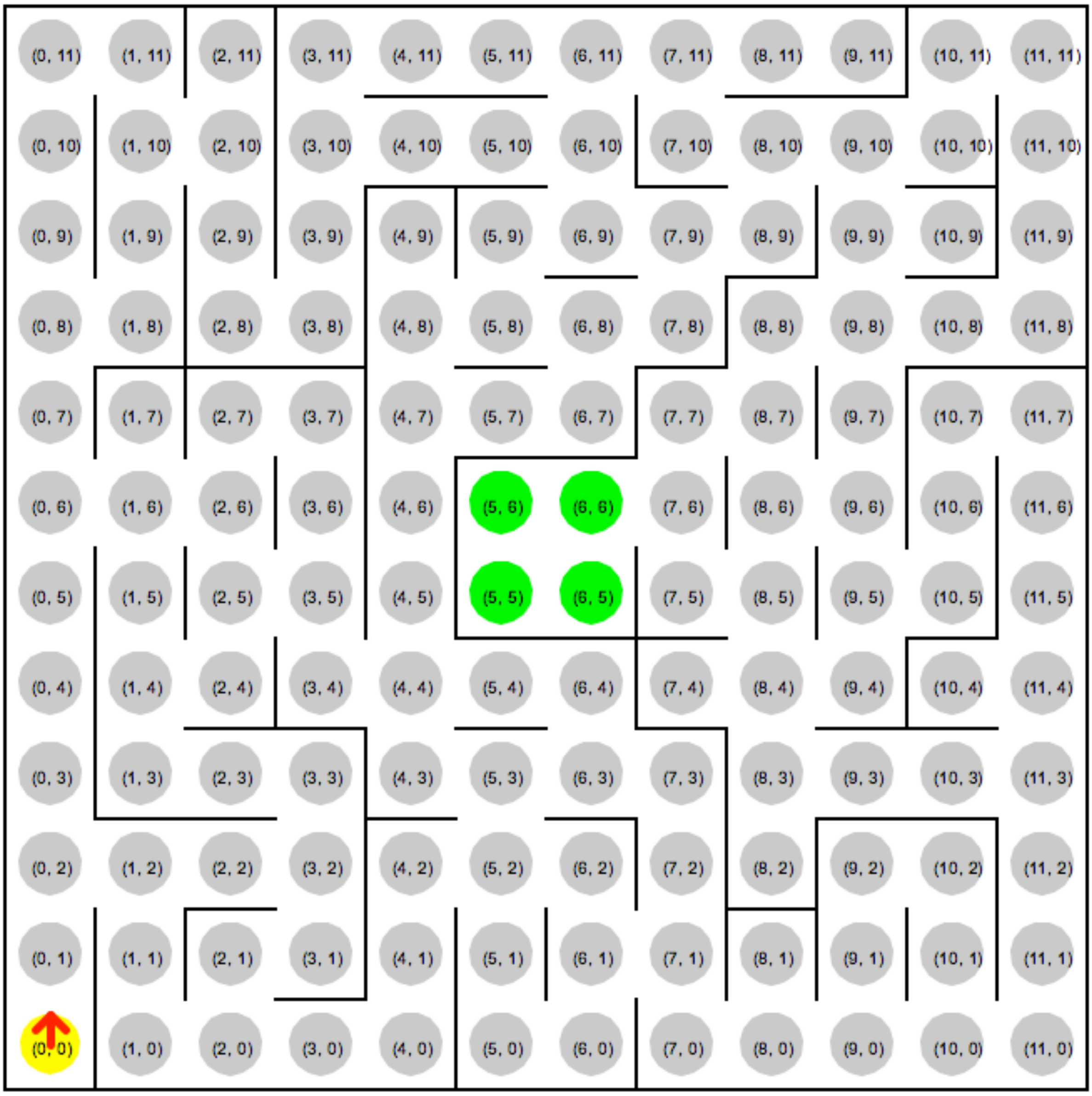
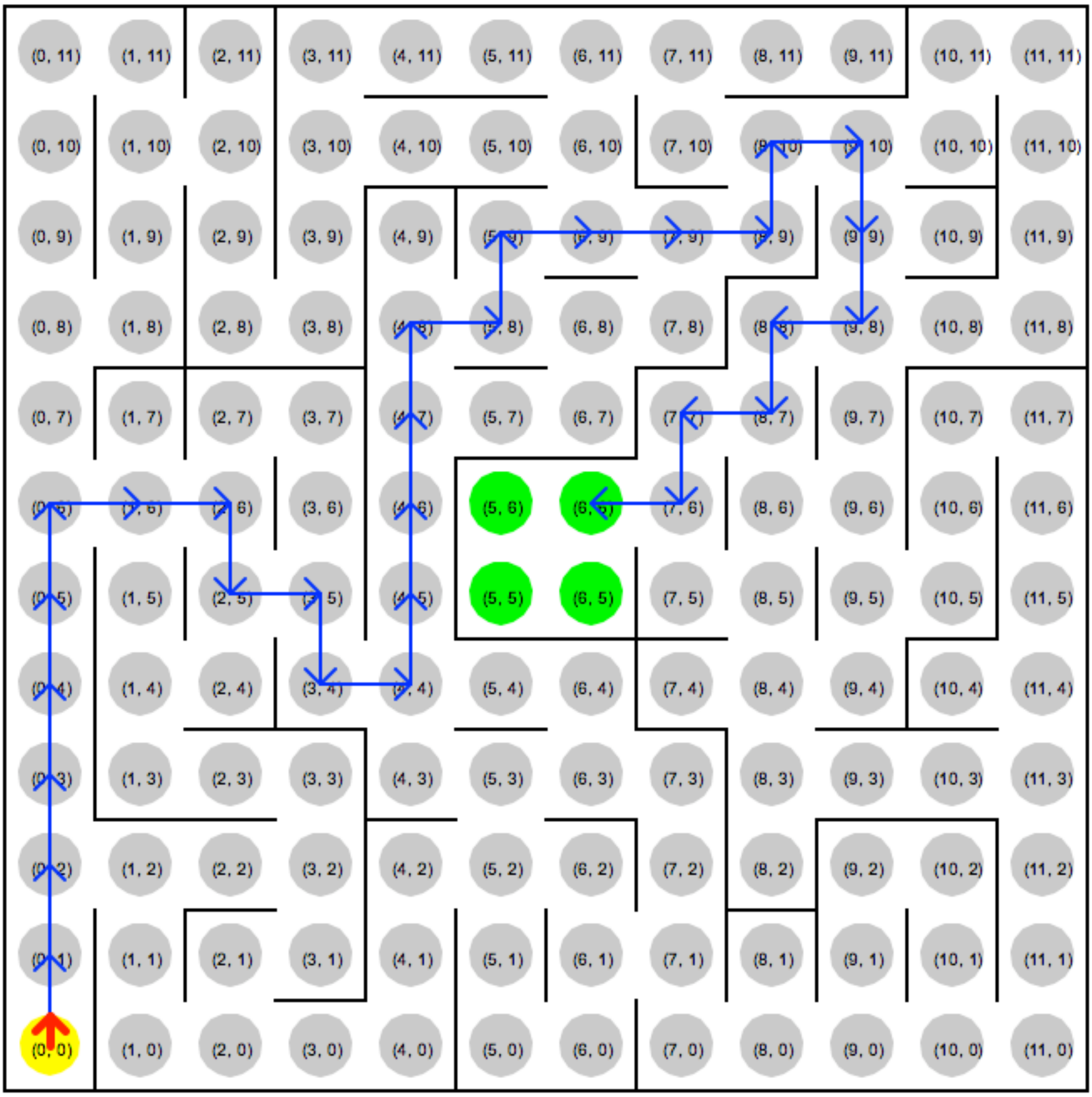
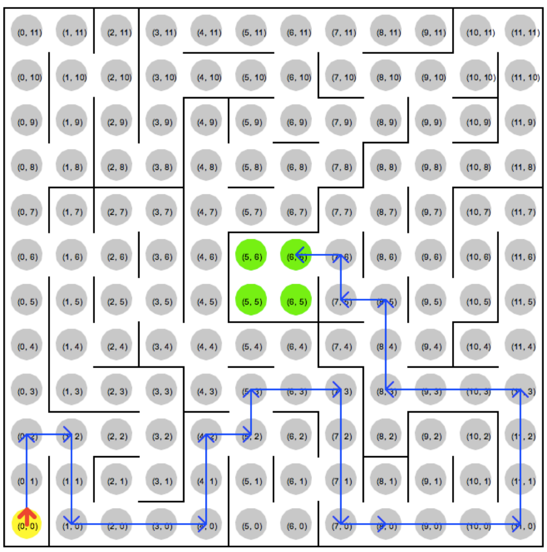
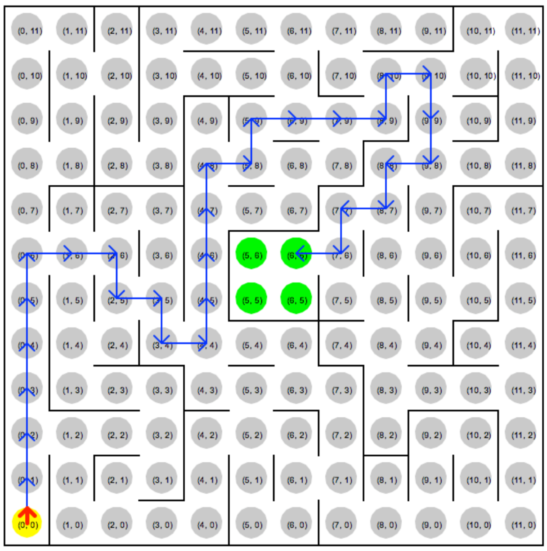


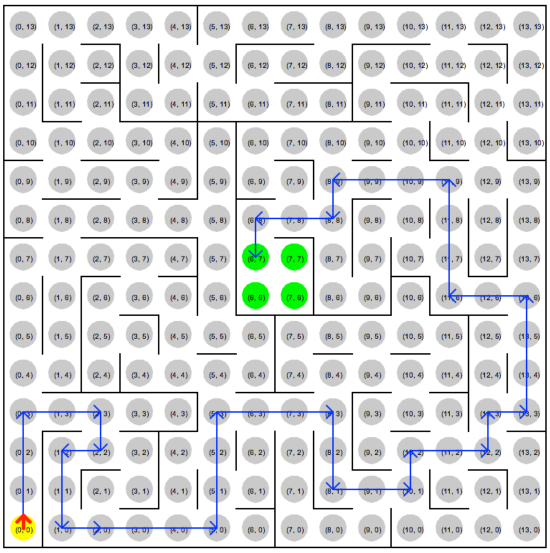
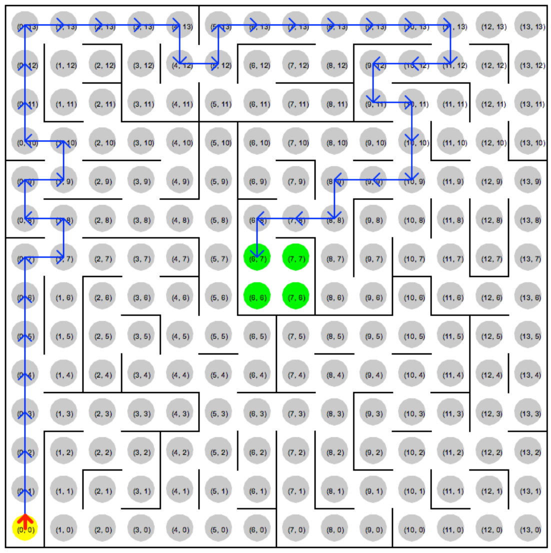
Illustration of single step shortest path.



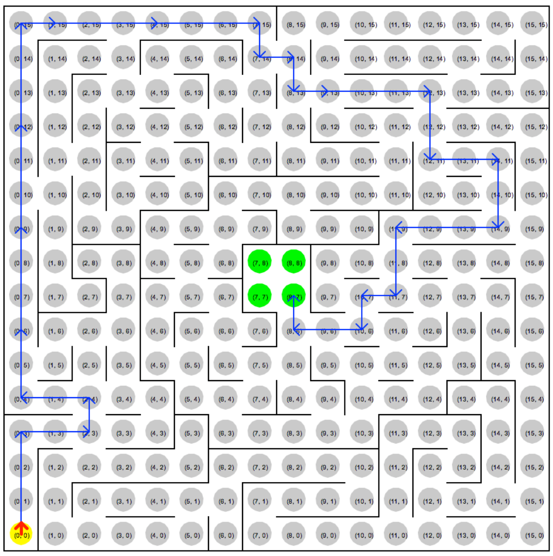
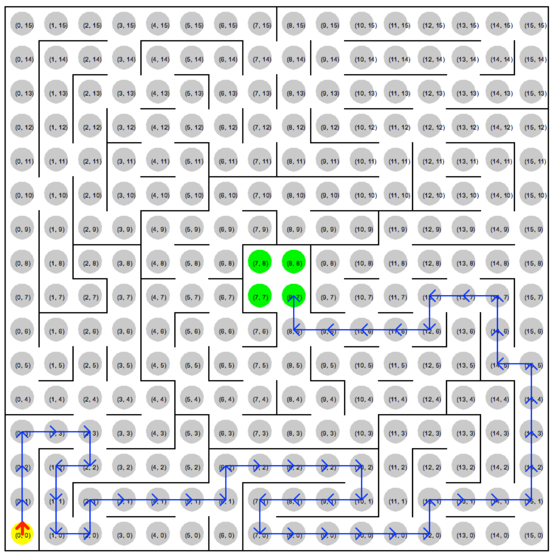
Because at most 3 steps are allowed, exploiting this parameter gives significant improvement

Multi Step: total 17 actions	Single Step: total 30 actions
	

Maze 2 (14x14)

Multi Step: total 22 actions	Single Step: total 43 actions
	

Maze 3 (16x16)

Multi Step: total 25 actions	Single Step: total 49 actions
	

Algorithms and Techniques

According to [Problem Statement section](), there are 3 stages: goal searching exploration, optimization exploration and second trial. In code implementation, I abstracted interfaces of these 3 stages so that they can be arbitrarily combined.

A. Strategies in goal searching exploration

The corresponding abstract class is `SearchingExploration` described in [Implementation section](#).

One-Step Random Turn

In a cell, a random turn with step size one is made. Because moving backward does not make mouse turn around direction, it's very likely that in next action it moves forward again. In order to decrease possibility of this going back and forth, only 3 actions are supported, namely, turning left, forward and turning right. However, there is also case where the mouse reaches dead end and hence the only way is to back off. In such case, turning right with no progress is returned.

Another optimization is that random guess is guaranteed not to include one step progress in the direction that hits a wall.

Strategy instantialization: `SearchingExploration_WeightedRandom()`.

One-Step Weighted Random Turn

Similar to One-Step Random Turn above, but weights of 3 actions can be customized to favor certain actions. For example, turn_weights=[2, 5, 2] means the weights of turning left, forward, turning right are 2, 5, and 2 respectively. Note that if action would lead to hit wall, the corresponding weight is would be zero, i.e. weighted random action is also guaranteed not to hit a wall.

Strategy instantialization: `SearchingExploration_WeightedRandom(turn_weights=[2, 5, 2])`

One-Step Favoring Unexplored Cell

Instead of random guess, action is determined by how much neighbouring cells are unexplored. By unexplored, we mean at least one edge of the cell is not identified. Use unexplored num as weight and then randomly guess one action.

One-Step Favoring Unexplored Space

An extended strategy of One-Step Favoring Unexplored Cell above. The idea is that although a neibouring cell status has been determined, that does not mean no information would be gained if that path is followed. Actually, the neighbouring cell is likely to lead to a cluster of cells that are unexplored. So the weight becomes the total sum of unexplored cell number one neighbour could lead to.

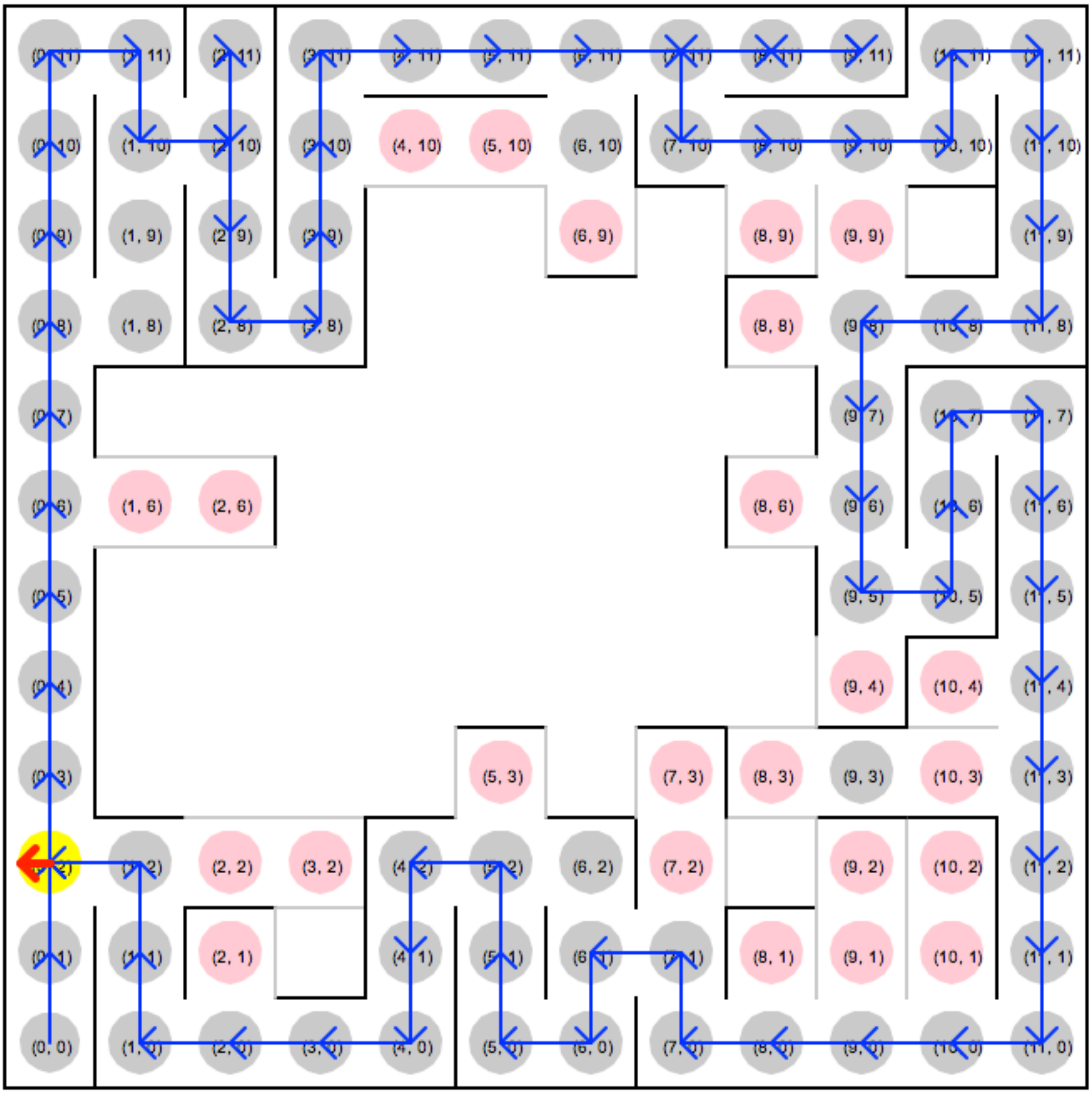
Strategy instantialization: `SearchingExploration_OneStepFavorUnexploredSpace()`

One-Step Wall Follower

Wall follower algorithm takes every left turn (left hand rule) possible but may get caught in loop. Further optimization is needed if it is guaranteed to reach goals.

Strategy instantialization: `SearchingExploration_LeftHandRule()`

Below shows a loop when applied to maze 1:



Multi-Step Goal Oriented (A*)

A* is an informed search or best-first search algorithm that generalizes Dijkstra shortest path algorithm. It employs heuristic function to assist it to find shortest path in most efficient way.

The cost function $f(n)$ is defined to be sum of actual distance and heuristic estimate of the distance from n to goal state.

$$f(n) = g(n) + h(n)$$

At certain cell k, we need to consider all reachable unexplored cells n_i and sort their $f(n)$ ascendingly and then take action to most promising (closest) one.

In my implementation, $g(n)$ and $h(n)$ are defined as follows

$g(n) = current\_steps + estimated\_steps(n, k)$

$h(n) = explored\_edge\_num(n) + manhattan\_dist(n)$

$explored\_edge\_num(n)$ in $h(n)$ means how many edge discovered, in attempt to favor unexplored cell under same manhatton distance.

Strategy instantiation: `SearchingExploration_GoalOriented()`

B. Strategies in second trial

The corresponding abstract class is `CalcShortestPath` described in Implementation section.

One-Step Dijkstra Shortest Path

Once entering 2nd run, a shortest path starting from (0, 0) with current maze status is computed using Dijkstra shortest path algorithm. This strategy ignores further sensor updates along the way and applies action series computed at initialization. Each action would make one progress.

Strategy instantiation: `DijkstraStride()`

Multi-Step Dijkstra Shortest Path

Similar to One-Step Dijkstra Shortest Path strategy but with max step size being 3.

Strategy instantiation: `DijkstraStride(max_step=3)`

Multi-Step Dijkstra Shortest Path with Sensor Updates

This strategy differes from previous one in that it considers sensor updates along the way and re-computes actions for each movement.

Strategy instantiation: `DijkstraStrideWithUpdates(max_step=3)`.

C. Strategies in optimization exploration

The corresponding abstract class is `ContinuingExploration` described in Implementation section.

No Optimization Exploration

Default strategy because enabling stage is not guaranteed to improve score.

Cover All Goals

After reaching first goal cell, it continues to reach other 3 goal cells if necessary. The intuition is to spend very restricted cost in hope to find a shortcut for run 2.

Strategy instantialization: `ContinuingExploration_CoverAllGoals()`

Coverage Threshold

Continue to reach unexplored cell until it covers enough cells parameterized by user.

With More AI

The intuition is that given information of currently explored maze, there may not be worthy to explore remaining cells, even though there are lots of them. For example, a cell serves a hole of unexplored cell cluster and the only entry is itself, and this fact can be deduced by current maze status.

Benchmark

The benchmark model strategy combination is **One-Step Weighted Random Turn** + **Multi-Step Dijkstra Shortest Path** + **No Optimization Exploration** and the result is given in Model Evaluation and Validation section. Because of stochastic nature, score is averaged across 5 trials. In case 1000 max step is exceeded, 100 would be the score.

III. Methodology

Data Preprocessing

No data preprocessing because the *test.py* script takes care of interaction between maze environment and the mouse. `Robot.next_move()` is the sole hook method required to implement.

Implementation

Implementation Principles

As Udacity machine learning capstone project, I want to achieve following implementation standards

- Readable: code is clearly documented and easy to be understood
- Composable: in order to test and compare different strategies of 3 phases identified in [Problem Statement section](), I abstracted each phase to one abstract class. Concrete strategy implements corresponding abstract class and is plugged into `Robot` constructor.
- Testable: those common routines such as manhattan distance, dijkstra shortest path algorithm, are separated out as standalone functions for test and resuable purposes. A dedicated unit test python script *robot_test.py* is provided.
- Diagnosable: Visualization of maze exploration status and its location and heading is implemented in *vis.py* script. In addition, I added defensive checks using Exception in several methods to enable quick failure so that I can corrrect my code as soon as possible.

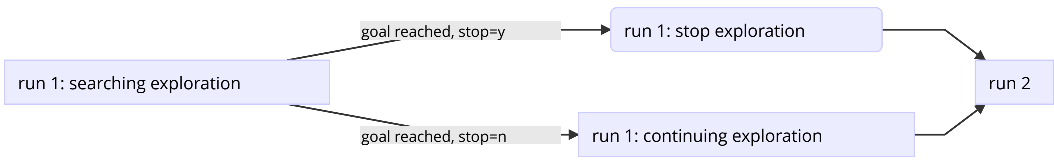
Skeleton of `Robot.next_move()`

`Robot.next_move()`, pseudo-code is given below:

1. Update `ExploredMaze` status by feeding it with sensor data.
2. Get action from corresponding strategy implementing class object.
3. Update robot status by calling `Robot.update_location(action)`.
    - `Robot.location`: new location after the action is executed
    - `Robot.heading` new heading after the action is executed
    - `Robot.trail` add the action to a list for route visualization
4. Switch to next stage if necessary.

Summary of Key Components

- `Robot.next_move(sensors)` defines 4 phases: run 1 searching exploration, run 1 goal reached and stop exploration, run 1 goal reached and continue exploration and run 2 phase. The phase transition graph is shown below.



- The central class is `ExploredMaze`, which keeps all information of the maze currently explored by the mouse. Everytime sensor information is fed in `Robot.next_move(sensors)`, `ExploredMaze` gets updated by via `ExploredMaze.sensor_update(loc, direction, depth)`.
- Underlying `ExploredMaze` lies `Cell` class, to which most methods of `ExploredMaze` are deligated. `Cell` keeps information of one grid of the maze such as which neighbouring grids have been explored and if yes, whether the edge of the cell leads to a wall or another cell.

- With all exploratary information kept in `ExploredMaze`, abstract class is defined for each phase (except run 1 goal reached and stop exploration phase), encapsulating common interfaces for plug-and-play strategy. These abstract classes are
- `SearchingExploration` abstract class encapsulates interfaces in goal searching exploration stage.
- `ContinuingExploration` abstract class encapsulates interfaces in optimization exploration stage.
- `CalcShortestPath` abstract class encapsulates interfaces in 2nd run.

Class `ExploredMaze`

Key methods of `ExploredMaze` are listed below:

```
class ExploredMaze(object):

    def sensor_update(self, loc, direction, depth):
        """
        Updates maze connectivity status with sensor information.

        Parameters
        ----------
        loc : tuple (int, int)
            Tuple of location, in format of (0, 0).
        direction : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.
        depth : int
            0 indicates wall;
            other positive number indicating how far from this `loc` mouse can
move forward.

        """

    def loc_of_neighbour(self, loc, direction, step=1):
        """
        Return location relative to `loc`. If the resulting location is out of
maze, None is returned.

        Parameters
        ----------
        loc : tuple (int, int)
            Tuple of location, in format of (0, 0).
        direction : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.
        step : int
            distance to `loc`, optional

        Returns
```

```
        -------
        new_location: tuple of (rotation: int, movement: int), or None
            None when out of maze.
        """

    def is_permissible(self, loc, direction, step=1):
        """
        Checks whether the mouse can move from a location along a direction
according to current connectivity status.

        Parameters
        ----------
        loc : tuple (int, int)
            Tuple of location, in format of (0, 0).
        direction : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.
        step : int
            distance to `loc`, optional

        Returns
        -------
        is_permissible: True, False, None
            None indicates there is not enough information along the way, i.e. not
sure there is wall between two cells in the way.
        """

    def compute_reachable_cells(self, loc_start=(0, 0), loc_excluded=None):
        """
        Computes set of cells that can be reached from (0, 0).

        Parameters
        ----------
        loc_start : tuple (int, int), default value (0, 0)
            Tuple of starting location, in format of (0, 0).
        loc_excluded : tuple (int, int), default value None
            Tuple of location to be excluded from path, in format of (0, 0).

        Returns
        -------
        reached_set: set of location tuple (int, int)
        """
```

Class `Cell`

```
class Cell(object):
```

```python
    def connect(self, direction, node):
        """
        Connects this cell with a neighbouring cell and vice versa.

        Parameters
        ----------
        direction : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.
        node : tuple (int, int)
            neighbouring location.

        Raises
        ------
        Exception
            When wall between these 2 nodes already established.
        """

    def set_wall(self, direction):
        """
        Sets a wall to one edge of current cell.

        Parameters
        ----------
        direction : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT, the direction relative to
current cell.

        Raises
        ------
        Exception
            When these 2 nodes are already connected.
        """

    def is_permissible(self, direction):
        """
        Checks whether the mouse can move one step from current cell along the
direction.

        Parameters
        ----------
        direction : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.

        Returns
        -------
        is_permissible: True, False, None
            None indicates there is not enough information, i.e. not sure there is
wall in that direction.
        """
```

```
```

Abstract Class `SearchingExploration`

```
class SearchingExploration(object):

    @abc.abstractmethod
    def next_move(self, loc, heading):
        """
        Decides next move according to `loc` and its `heading`.

        Parameters
        ----------
        loc : tuple (int, int)
            Tuple of location, in format of (0, 0).
        heading : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.

        Returns
        -------
        next_action : tuple of (rotation: int, movement: int), or tuple of
('Reset', 'Reset')
            For example: (90, 3).
        """
        pass
```

Abstract Class `ContinuingExploration`

```
class ContinuingExploration(object):

    @abc.abstractmethod
    def next_move(self, loc, heading, steps):
        """
        Decides next move according to `loc` and its `heading`.

        Parameters
        ----------
        loc : tuple (int, int)
            Tuple of location, in format of (0, 0).
        heading : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.
        steps : int
```

```
            number of steps

        Returns
        -------
        next_action : tuple of (rotation: int, movement: int), or tuple of
('Reset', 'Reset')
            For example: (90, 3).
        """
        pass
```

Abstract Class `CalcShortestPath`

```
class CalcShortestPath(object):

    @abc.abstractmethod
    def compute_p2p_action(self, loc_start, heading, *args):
        """
        Computes list of actions that would follow shortest path starting
`loc_start` with `heading`.

        Parameters
        ----------
        loc_start : tuple (int, int)
            Tuple of location, in format of (0, 0).
        heading : int
            Must be D_DOWN, D_LEFT, D_UP, D_RIGHT.

        Returns
        -------
        action_list : list of tuple (rotation: int, movement: int)
            For example: [(90, 3), [0, 3)]
        """
        pass

    @abc.abstractmethod
    def next_action(self):
        """
        Returns next action computed previously by `compute_p2p_action`.

        Returns
        -------
        action : tuple of (rotation: int, movement: int)
        """
        pass
```

Refinement

A* algorithm servers the core of the project but lots of detailed debugging issues and code refactoring or design pattern cost much more time.

One typical debugging example is due to the fact that moving backword does not make the mouse change heading. In initial implementation of `SearchingExploration_OneStepWeightedRandom.next_move()`, weight of moving backward is supported but it caused problem of excessing 1000 step limit. Until visualization with heading is provided is issue identified and resolved.

Another typical design pattern issue is that I always need to move methods into appropriate class so that the interactions between components are minimized and reasonable.

IV. Results

Model Evaluation and Validation

Combined Strategy	**Maze 1**	**Maze 2**	**Maze 3**
Benchmark One-Step Random Turn Multi-Step Dijkstra Shortest Path *No Optimization Exploration*	50.99	58.65	62.91
One-Step Favoring Unexplored Space Multi-Step Dijkstra Shortest Path *No Continuing Exploration*	77.79	89.03	100
One-Step Weighted Random Turn (weights=[3, 5, 3]) Multi-Step Dijkstra Shortest Path *No Optimization Exploration*	42.38	78.34	42.59
Multi-Step Goal Oriented (A*) One-Step Dijkstra Shortest Path *No Continuing Exploration*	32.033	48.667	56.600
Multi-Step Goal Oriented (A*) Multi-Step Dijkstra Shortest Path *No Continuing Exploration*	**19.033**	**27.667**	**32.60**
Multi-Step Goal Oriented (A*) Multi-Step Dijkstra Shortest Path with Sensor Updates *No Continuing Exploration*	19.033	27.667	32.60
Multi-Step Goal Oriented (A*) Multi-Step Dijkstra Shortest Path *Cover All Goals*	19.100	27.733	32.667

Note that score with underscore means the result is stochastic, and is averaged across 5 trials. If a trial exceeds 1000 step in exploration run, 100 is given as penalty score. The concrete trial scores are listed below.

Combined Strategy	Maze 1	Maze 2	Maze 3
Benchmark One-Step Random Turn Multi-Step Dijkstra Shortest Path *No Optimization Exploration*	39.33 22.23 100 49.33 44.07	100 35.667 100 29.633 27.967	100 100 46.667 35.267 32.633
One-Step Favoring Unexplored Space Multi-Step Dijkstra Shortest Path *No Continuing Exploration*	43.367 100 45.567 100 100	100 45.167 100 100 100	100 100 100 100 100
One-Step Weighted Random Turn (weights=[3, 5, 3]) Multi-Step Dijkstra Shortest Path *No Optimization Exploration*	37.433 28.133 24.1 100 22.233	42.8 100 48.9 100 100	30.733 31.133 42.4 53.367 55.333

Justification

Combined Strategy	Maze 1	Maze 2	Maze 3
Benchmark One-Step Random Turn Multi-Step Dijkstra Shortest Path *No Optimization Exploration*	50.99	58.65	62.91
Multi-Step Goal Oriented (A*) Multi-Step Dijkstra Shortest Path *No Continuing Exploration*	**19.033**	**27.667**	**32.60**

The best model outperformed the benchmark model substaintially. And due to its deterministic nature, the result can always be reproduced. Actually, running `robot.py` without modifications would arrive at these scores.

V. Conclusion

Free-Form Visualization

Reflection

I am particularly interested in theoretic details of A* algorithm such as admissible and consistent properties. In addition, I start to get to know more sophisticated search algorithms

- Incremental Replanning Algorithm, e.g. D* and D* Lite
- Anytime Algorithm, e.g. ARA*
- Anytime Replanning Algorithm, e.g. AD*

Improvement

There are several things in my opinion that can get improved.

1. Implement coverage threshold continuing exploration strategy and see if there is score increase or decrease.
2. In `SearchingExploration_GoalOriented` and `DijkstraStrideWithUpdates` , `dijkstra_shortest_path()` function is called several times. This is very inefficient because there are few status updates between successive calls. A family of dedicated algorithms exist, most notably D* Lite algorithm would help a lot.
3. Identify several typical cases in optimization exploration stage, such as what is mentioned in **With More AI** so that mouse is encouraged to continue exploring when finding shortcut opportunity is high but discouraged vice versa.
4. Currently, visualization module (*vis.py*) using Turtle library is slow, partial update can be employed to accerelate rendering process by providing live action.